

# **Safety Software Design for Microcontrollers**

## **Part 1: Problems of a Typical Software Design for Microcontrollers**

### **ABSTRACT:**

This part of the course and all the following parts are aimed at the software engineer who wants to enhance his skills regarding microcontroller software design. The pre-requisite is that you have a good knowledge in C-programming and a good understanding of microcontroller functionality.

Part 1 of the course will point out the problems of a typical software design like it is very often performed for microcontrollers. It will show you that it is not enough to have the complete functional requirements for a task. You still can make errors in the design which will lead to problems in the progress of the design and coding phase itself, but also regarding testing and later maintenance.

Possible solutions will be pointed out for each discussed design error. However these hints are only outlines and summaries of solutions which will be discussed in much more detail in the subsequent parts of the course.

Copyright  
Eberhard De Wille  
Zieglerweg 4  
92287 Schmidmuehlen  
Germany

**Table of contents**

1. Problems of a Typical Software Design for Microcontrollers.....	4
1.1. Introduction.....	4
1.2. Why is Software Design for Microcontrollers different? .....	4
1.3. An Example of a Typical Design.....	6
1.3.1. Loss of Portability in an Ad-hoc Design.....	11
1.3.2. Bad Maintainability in an Ad-hoc Design .....	13
1.3.3. Global Variables lead to problematic Interfaces.....	14
1.3.4. Global Variables lead to a problematic Data Flow and Module Split Up .....	17
1.3.5. Other possible Pitfalls of Global Variables.....	23
1.3.5.1. Unwanted Cross Influences .....	23
1.3.5.2. Change of Linkage .....	24
1.3.6. The use of Basic Data Types.....	27

## **1. Problems of a Typical Software Design for Microcontrollers**

### **1.1. Introduction**

Microcontroller programming is a world of its own. The number of applications has grown considerably during the past years, the number of available microcontrollers has exploded and the modern world would not be the same without microcontroller applications. Twenty years ago microcontroller applications were limited to a few controls in aircraft and a few other sophisticated applications as for example in vehicles. Today even your coffee maker may contain a microcontroller and there is no vehicle on our roads which does not at least contain one for the engine control. Some high-end vehicles even contain 30 and more microcontroller systems and the Airbus family starting with the A320 flies completely based on microcontrollers, without hydraulic backup systems.

However the programming of these systems did not change a whole lot compared to a system 20 years ago. The programs became bigger, yes, but the way of design is not any better. Only low volume highly safety critical systems such as aircraft and nuclear power plants are constructed using better development methods. For these systems considerable steps towards the better were made during the past decades. However, these systems are not in focus of this training. For normal applications the state of the art is more or less the same as it has been 20 years ago. Only where eager software architects, quality engineers and design specialists try to enforce improved ways of design and programming you may see some improvement. As size and complexity of microcontroller programs will grow even further in near future it is absolutely necessary to improve this situation. This course is aimed at the microcontroller programmer, to help him to achieve this goal. Therefore this course is strongly biased towards microcontroller applications. This was a deliberate decision. There are excellent courses, material and literature which help programmers to improve their design for normal computing applications. The lack of sufficient literature and material for microcontrollers shall be filled by this course, at least to some degree.

### **1.2. Why is Software Design for Microcontrollers different?**

Software design for microcontrollers is very different from design for other applications as for example programs for the PC world. This situation is mainly because there are some project conditions common to most microcontroller projects which lead to a typical design. The most important one of these conditions are the following:

1. Microcontroller applications usually have an additional hardware dimension to cope with. Where as for the normal programming e.g. of the PC the hardware platform is stable and known, for microcontroller applications the hardware platform is usually new and very often unsettled. If you have to set up a new microcontroller software this will be in many cases on a new hardware. Even if the CPU is known from other projects, there will be other circuits in the system which are new and need attention. This may range from issues of the CPU core as e.g. a new external memory, a new clock frequency to make the system faster, to new external hardware features at the inputs or outputs which

require the attention of the programmer. Based on this situation the microcontroller programmer has to work on two things. First of all the hardware and secondly the software he has to provide for it. The first approaches will therefore usually neglect software design rules, postpone them to later stages of development, and rather attempt to set up the expected functionality of the system. In this stage even hardware adaptations may be needed and new methods have to be found to drive outputs or evaluate inputs.

2. Applications for microcontrollers are sometimes very small. At least this is true for the first samples of a new product. In this stage of a project many programmers think that it is easy to keep an overview over the code and that employing proper design would constitute an unnecessary overhead. However they often forget that applications will grow and eventually the overview may be lost and they run into the design problems outlined in this chapter.
3. Microcontroller applications need extensive debugging and analysis possibilities. In most applications it is necessary to be able to have snap shots of the program data of every execution cycle. There are standard tools which provide these features but usually they require that all variables which have to be watched are global variables. Thus external tooling is influencing the way of design for the microcontroller application. There are ways to deal with this, and some of the possibilities will be outlined in this course.
4. Microcontroller programming is usually still done in C without employment of other design tools or methods. Even the use of object oriented languages like C++ or Java is not gaining much ground. The reason is simply money. Since microcontroller applications are usually produced in high quantities it is important to save even a single cent on the hardware, which means that there is a high pressure to save RAM and ROM resources. All advanced design methods and their related tools however, will still result in overhead concerning the ROM and RAM size of the generated programs, as well as in an increase of their runtime. Driven by the need to fulfill resource constraints a certain programming style evolved in the microcontroller programmer community. The details of this style will be outlined in the following part of this chapter.



**Reasons for a problematic design are:**

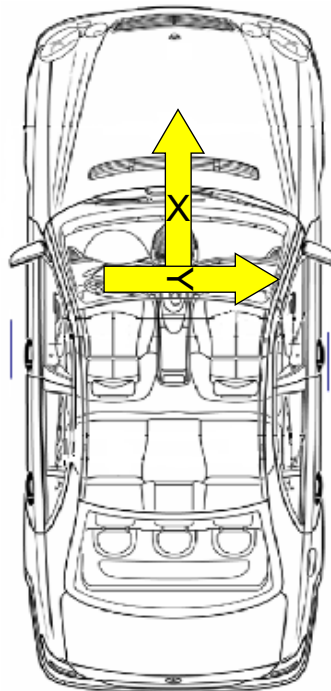
- **The additional hardware dimension in a project defers the attention from software design issues.**
- **There is a need for global variables because of debugging and measurement.**
- **The aim is to save RAM and ROM because this costs money in each unit which is sold. A good design is considered to generate overhead and thus additional costs.**

### 1.3. An Example of a Typical Design

Having outlined the specific project setting in which microcontroller programming usually is performed I want now to go into the details of this programming style which evolved from these environmental conditions. To do this I want to use a small example which shall accompany us throughout the whole course. Like it is usually done in software development I want to outline the requirements of this software before going into the actual design or implementation. The sample design shall be for a very simple airbag software:

#### **General Description of the Software:**

The SW design of a simple airbag software shall be performed. An airbag system for a front airbag i.e. the one protecting the driver in a car is based on a acceleration sensor mounted in a way that it detects accelerations in the driving direction, the so called x - direction.



If a car crashes into a barrier this sensor will generate a very high signal for a couple of milliseconds because by crashing into the barrier the car, which was driving before on high speed, is virtually stopped and will start to deform to consume the energy. This event can be clearly differentiated from other driving events such as a strong braking by the amplitude of a signal and it's duration.

Because you would hate to have an airbag fired on a sensor problem, such as a shortcut to a supply voltage, we need a plausibility for the main sensor. A shortcut may appear to the

detecting hardware circuits similar to a real signal, but since it is unlikely that a shortcut would appear on a second sensor at the same time, a second sensor would be needed to be able to check on the main sensor. However, sensors are expensive and therefore a certain method is used in common airbag systems which also contain side airbags. The side airbags give protection to the driver in case another car crashes into the driver's door. To detect this the same principle is used as for the front airbag and another sensor is mounted turned around by 90 degrees, the so called y- direction, to provide optimized sensing in the direction looking towards the doors of the car. However, the shock waves of acceleration are always detected by all sensors in the car, no matter which kind of crash it is. The only difference is that if a sensor detects an acceleration which is out of his sensing direction, the signal will be considerably smaller. Nevertheless the signal can be used to check the plausibility of the main sensor.

Now let us go into the description of the software which shall be generated. It shall contain all necessary software parts to enable the operation of the software on a C164 microcontroller. This means it shall consist of the hardware relevant code portions as well as an algorithm. The algorithm shall provide front crash detection via the so called X-sensor. This sensor measures the acceleration of the vehicle in driving direction. This signal has to be safeguarded by a plausibility functionality using the so called Y-sensor of the side crash detection. This side crash detection shall not be implemented here. Further in this example the firing decision shall trigger a belt pretensioner first, and after a 5ms time delay it shall trigger the front airbag. The belt pretensioner shall be only fired if the buckle switch is activated. That means that if the driver is unbelted the airbag shall be fired immediately without waiting for a belt pretensioner timing. The general firing decision for the pretensioner and airbag shall be made if the 5ms averaged X-signal is exceeding a certain threshold.

### **The Operating System:**

It is assumed that the operating system is present. It shall not be programmed in the progress of this exercise, however the functions which will be called by the operating system in certain periods of time shall be part of the exercise. There is an AD converter interrupt which will be executed every 250us. After the initial two 250us interrupts the operating system will call immediately a 500us task and after each second 500us task there will be the call of a 1ms task. It can be assumed that the calling of the interrupts and tasks is stable, so that no extra measures have to be taken concerning the operating system. This means that we have the following functions:

1. A function called every 250us (by an interrupt).
2. A function called every 500us (by the operating system).
3. A function called every 1ms (by the operating system).

The sequence of the tasks shall be:

--2-----2-5-----2-----2-5-1----

2 = 250us task  
5 = 500us task  
1 = 1ms task

### **The Analog Signal processing:**

A 10-bit analog to digital converted value will be provided for both the X and the Y sensor. The X sensor has a sensitivity of 56.16mV/g and range of -30g to +30g. The Y sensor has a sensitivity of 27.57mV/g and a range of -70g to +70g. The X sensor is available on analog channel 0 and the Y sensor is located on analog channel 1 of the C164 microcontroller. The AD signals will be sampled by the AD converter every 250us in a dedicated interrupt. For each signal channel an averaging has to be done every 500us i.e. after 2 AD samples. In the 1ms task another averaging shall take place to build an average on two 500us averages.

### **The Digital Signal processing:**

There is a buckle switch on the system which detects if the driver is belted or unbelted. The switch will be read on the port 3 pin 18 of the C164 microcontroller. The buckle switch has to be de-bounced by a 30ms de-bounce time to generate a stable switch signal.

### **The Detection Algorithm:**

The crash detection algorithm shall be called every 1ms. If the average of the last 5 values of the X-sensor is above the threshold of 18.7g the firing shall be triggered. The firing has to be safeguarded by a threshold of the Y-signal. Its average of the past 5 values has to exceed the value of 4.6g.

### **The Fire Logic:**

The next stage after the detection algorithm is a fire logic. The fire logic has to trigger two events upon the detection of a fire condition. In modern airbag systems there is usually a so called belt pre-tensioner. This device will tighten the safety belt and pull the driver back into the seat before the airbag is fired. The belt pre-tensioner has to be fired first, but only if the buckle switch is activated. In this case the front airbag shall be fired after 5 ms delay time, to allow some operating time to the belt pre-tensioner. In case the buckle switch is not activated the airbag shall be fired immediately.

You will find a full code sample for this specification in the supplementary material of this course. It shows how many programmers would implement the code after reading this specification. The usual approach in designing this software would be to open a software module and first of all make a number of functions. Which functions these are, depend on the experience of the programmer who can estimate what could be best placed in a subfunction and what is small enough to remain a piece of code in one of the operating system tasks. Although the number, names and size of functions may vary during the development of the software, it may look like the following functions in a single module:

**1 Software Module**

```
void MainAlgorithm (void)
```

```
void FireLogic (void)
```

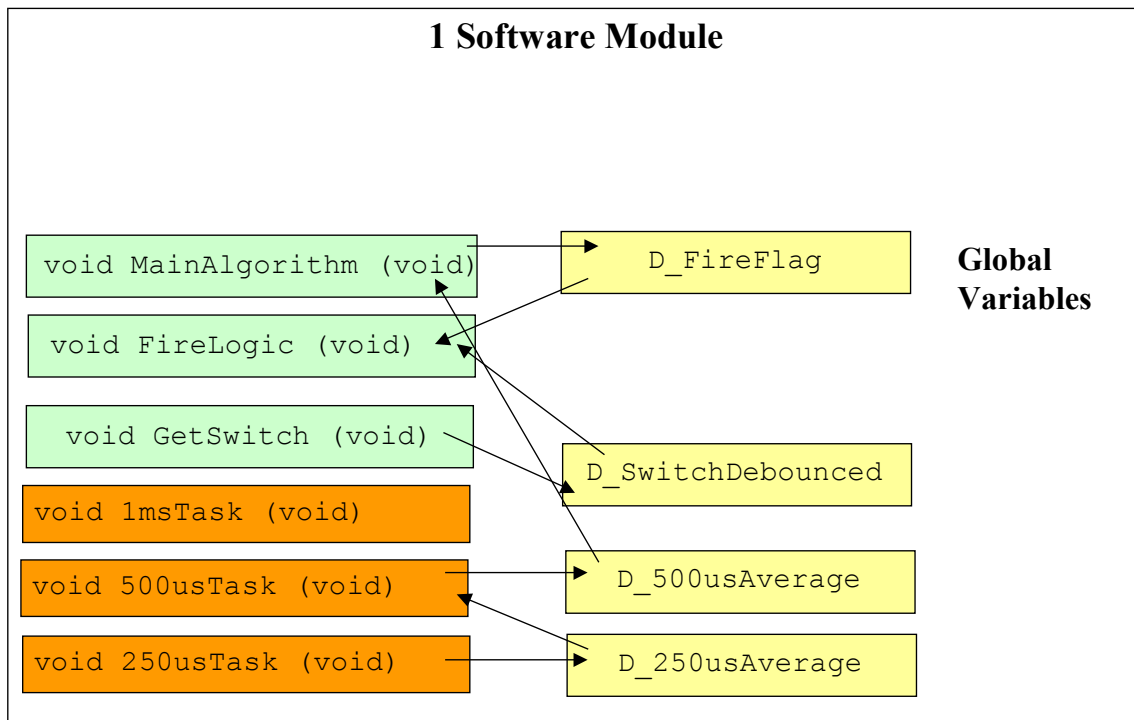
```
void GetSwitch (void)
```

```
void 1msTask (void)
```

```
void 500usTask (void)
```

```
void 250usTask (void)
```

The exact interface to the individual functions is hard to determine at the beginning. Therefore the functions would be typically designed as "void" functions, without pass parameters (also called function arguments), and without a return value. Further, it is commonly believed in the programmer community that pass parameters are an overhead and the use of global variables is the most effective way to use data. As the code in the functions grows the data are evolving step by step in parallel. This leads to a number of global variables which contain the program's data. The access to these data by the various functions is indicated by the arrows in the following picture:



Based on the outlined specification I implemented a sample code in the typical way of microcontroller software design. The complete code is contained in a file called "**airbag-bad-design.zip**" and can be downloaded as part of the training.

On the first glance this code looks very nice and neat. Since it is very small it is easy to keep an overview over the complete software. You even find some subfunctions which contain a certain functionality, although the lines in the subfunctions could have been just as well copied into the main function to build a linear code. It seems that there is no overhead and this way of design is the best what you can get. Well, I agree for exactly **this** little piece of code. It is only an example and therefore I tried to keep it small. But imagine that the application is bigger and more complex. Imagine that there are various software engineers working on the project. Let's say 5 engineers working on 30 modules with approximately netto 1500 lines of code for each module. This is still a small project but imagine the impact and the cross influences on the complete project when I discuss the possible problems now.

I will pick out now some outstanding portions of this code to illustrate the problems of a typical ad-hoc design.

### 1.3.1. Loss of Portability in an Ad-hoc Design

Microcontrollers have a number of peripherals which need to be accessed by the program. Some microcontroller makers provide a set of definitions e.g. structs which are mapped over the set of peripheral device registers. Thus the registers can be accessed like normal global variables. For other microcontrollers the compilers contain a set of proprietary commands to access the peripheral devices. The following code portion is part of the switch evaluation function of our sample software, which debounces the switch information:

```
void GetSwitch()
{
    T_UBYTE ub_SwitchLevelRaw;
    /* debounce switch signal */
    ub_SwitchLevelRaw = _getbit(3,16);
    if (ub_SwitchLevelRaw != ub_PreviousSwitchLevel)
    ....
```

What you find in there is a non-standard C instruction. The "**\_getbit**" is a proprietary C164 compiler instruction to read a certain port pin. This instruction is even proprietary to a certain compiler maker. There are other compilers for the C164 where this instruction looks different.

Another example in our code is the analog to digital conversion as it can be seen in the following code portion:

```
void Task250us()
{
    /* start the requested conversion for channel 0 */
    ADCON = 0 | 0x0110;
    /* wait until the conversion is finished */
    do { ; } while (ADBSY);
    uwa_Sensor1[ub_BufferToggle] = ADDAT;
    ...
```

In this sample we have the 250us task which is e.g. directly called by an interrupt service routine or the operating system. This task contains direct accesses to the AD-converter registers as e.g. "**ADCON**" or "**ADDAT**". Although the code seems to have an optimized size and no overhead several problems arise from this kind of design.

1. You loose portability. In case you have to use a different microcontroller you have to change a lot of places in the code. Even if you stay with the same microcontroller and switch to another compiler maker you will run into this problem. You have to touch each module which contains such compiler specific syntax. This also means that you have to re-test the complete software.
2. You loose even portability within the same microcontroller family. In case you want to switch to a next bigger controller of the same family your adaptations may take you through the complete code, depending on the compatibility of the new controller to the old one. Or imagine some hardware changes from one version to the next version of your samples, even without a controller change. E.g. if the hardware engineer has to swap some port pins due to radio interference problems. Then you end up going through the complete code for changes, which is especially true if e.g. the same registers and port pins are accessed multiple times at different places of the code. You easily overlook

the one or the other of these places. There is no good mechanism to avoid this. For swapped port pins both pins are still valid, but accessed incorrect and you end up with functional misbehaviour of your system. A misbehaviour like this is hard to detect.

3. Portability is not only concerned with the target hardware platform. Portability also means to be able to run a code in a standard testing environment. This is not possible if you use proprietary syntax. If there is virtually no source code modules without proprietary code lines, you are forced to do all your testing in the original hardware with an emulator or debugger. You will not be able to employ standard test tools with their mighty script languages. These tools would improve regression testing and test automation and even automatic test case generation and coverage measurement. This world is closed to you and you have to rely entirely on the tooling which an emulator or debugger maker will provide for you. In most cases these tools are very expensive and have a lot of restrictions. One of these restrictions is e.g. the upload time of program code. Personally I did a lot of programming on the MPC563. The tool used for testing and debugging was a standard J-TAG debugger. If you want to test or debug a program you have to upload it into the FLASH memory first. This usually takes several minutes. A typical value would be 5-7 minutes for a standard application. This means that for every little code change e.g. to try out something, to come closer to the problem you are looking for, you have an average of 2 minutes to make the code change and compile and then you sit there waiting for the download for 7 minutes. In other words the overhead a wrong testing environment and a bad software design can give you easily reaches 400% and more. In other words for work which you could easily do in 2 hours you will take a complete day.

A common way to avoid these problems is to use macros instead of making direct use of proprietary compiler functionalities. However this involves other problems. Function like macros should be completely avoided because of the involved problems as they are discussed in the "safe coding in C" course. Further the use of macros usually does not take care of the sequences which may be needed to control a peripheral device. In the above example it is e.g. necessary to load the "ADCON" register before polling the "ADBSY" flag. In another controller environment the AD-converter handling may look completely different and a macro hiding this design decision may be completely obsolete and can not be adapted to a new environment. The best solution to solve this is to introduce a hardware abstraction layer. This solution will be outlined later in the course.



**Reasons for a problematic design are:**

- **The use of implementation (compiler / CPU) specific commands in the source code leads to the loss of portability. It also leads to a bad maintainability if the same commands to access the same peripherals are used multiple times in the code. The testability also suffers greatly. You can not use standard test tools and**

**automation, but you have to rely on the tools and methods provided by the hardware makers.**

**Possible Solution:** A possible solution to avoid this design problem is the introduction of a Hardware Abstraction Layer as outlined in the architecture part of this design course. This will enable you to generate the biggest part of your code independent of the CPU hardware. You can use standard test and design environments to a large degree. The proprietary syntax will be kept in specific modules which can be tested separately on the debugger and target hardware.

### 1.3.2. Bad Maintainability in an Ad-hoc Design

If you look at the following code portion you detect another design sin which is typically performed in ad-hoc controller programming:

```
void MainAlgorithm()
{
    T_SWORD sw_XSignal;
    T_SWORD sw_YSignal;

    /* detection algorithm code */
    sw_XSignal = (T_SWORD)((((T_SLONG)uw_Sensor1_lms - ZERO_POINT) * SENSOR_X_FACT) >> 5);
    sw_YSignal = (T_SWORD)((((T_SLONG)uw_Sensor2_lms - ZERO_POINT) * SENSOR_Y_FACT) >> 5);

    if ( sw_XSignal > 187)
    {
        uw_MainSignalCounter++;
    }
    else
    {
        uw_MainSignalCounter = 0;
    }
    if ( sw_YSignal > 46)
    ...
}
```

Whenever a threshold is used in the software, or a counter value is checked the tendency is to put them as direct constant numbers into the code. If these numbers are not used a second time somewhere else in the code there is no reason to change this later on. Even if the same number is needed another time later on, it would need some extra thought and effort to take it out of the code into a `#define`. However, if from a functional aspect the same constant value is used multiple times in the code, any maintenance on it will become a nightmare. You may change the constant at one place but overlook the other place. This will lead to strange functional behaviour and is hard to detect.



**Reasons for a problematic design are:**

- **The use of hard coded constant values in the source code leads to the decrease of maintainability. If the same constant from a functional aspect is used multiple times in the code it leads also to an increased liability to errors.**

**Possible Solution:** Constants should be defined in a header file which is included by all software modules of the project.

### 1.3.3. Global Variables lead to problematic Interfaces

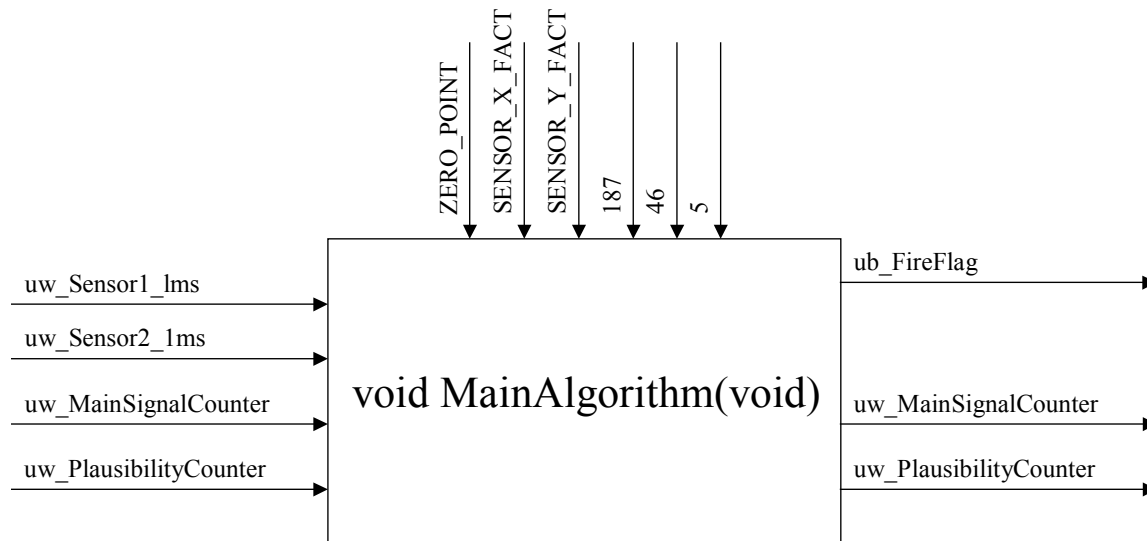
One of the main reasons for using global variables instead of the technical interface of the C-functions is that many functions have the need to return more than one value as an output. The return values of C-function are meant to return a single value, which is not enough in many cases. There are solutions which provide a work around for this, but they will be discussed later on in the course.

The following function is typical for most functions in microcontroller programming. As you see the technical interface is not used. The function is defined "void" in it's pass parameter list and also "void" for it's return value. So where is the interface to the function?

```
void MainAlgorithm()
{
    T_SWORD sw_XSignal;
    T_SWORD sw_YSignal;

    /* detection algorithm code */
    sw_XSignal = (T_SWORD)((((T_SLONG)uw_Sensor1_lms - ZERO_POINT) * SENSOR_X_FACT) >> 5);
    sw_YSignal = (T_SWORD)((((T_SLONG)uw_Sensor2_lms - ZERO_POINT) * SENSOR_Y_FACT) >> 5);
    if ( sw_XSignal > 187)
    {
        uw_MainSignalCounter++;
    }
    else
    {
        uw_MainSignalCounter = 0;
    }
    if ( sw_YSignal > 46)
    {
        uw_PlausibilityCounter++;
    }
    else
    {
        uw_PlausibilityCounter = 0;
    }
    if ((uw_PlausibilityCounter >= 5) && (uw_MainSignalCounter >= 5)) ub_FireFlag = 1;
}
```

On a first glance it is not easy to comprehend which are the inputs to the function, which are the outputs and what constants are used. It needs a very close look into the function to detect the actual interface. The following drawing visualizes the interface and the constants to the function:



Two variables are only an input, one variable is only an output and two counter variables are read and written by the function. In my example these are only small functions where you have a good chance to comprehend everything fairly easy. However, in my testing activities I came across functions with more than 40 input variables!! Using the pass parameter interface would set certain limit to the size of the interface. The compiler would allow only a certain number of parameters and beyond that the system performance would be bad if a pass parameter list is too big. Most CPUs work best with a maximum of 4 to 8 pass parameters. Beyond this value the performance decreases, because the values have to be stored on the stack if not enough registers are available. Of course such an excessiv "interface" of 40 global variables originates first of all in a much bigger design problem, but doing maintenance or testing you would not be allowed to change such a function. Instead you would have to struggle with it's interface. First of all, if you have that many inputs you simply loose oversight. The function will be far too big. Most likely it could be split into several individual functions. Further a function with this kind of interface is not testable any more. Some test strategies require to have test cases with the minimum and maximum value of each input then you need to combine the inputs and run each input with all the possible combination of the other inputs. I saw something like this running for two weeks on a powerful machine and we had to stop it because calculations showed that it would run for some more weeks. Something like this is simply not testable!



Reasons for a problematic design are:

- The use of global variables tends to generate functions defined as void - void. This means that maintainability and testability is reduced or even lost. The overview over the inputs and outputs of the function is not easy to comprehend and functions tend to have no limits regarding the size of the interface.

**Possible Solution:** Simply use the technical interface of the programming language. Having void functions should be an exception. If you use the technical interface the size of the pass parameter list automatically will force you to rethink your function size and content and split functions earlier. The above example could be implemented like this:

```
T_UBYTE MainAlgorithm(const T_UWORD uw_Sensor1_lms, const T_UWORD uw_Sensor2_lms,
                    T_UWORD * puw_MainSignalCounter, T_UWORD * puw_PlausibilityCounter)
{
    T_UBYTE ub_FireFlag;
    T_SWORD sw_XSignal;
    T_SWORD sw_YSignal;

    /* detection algorithm code */
    ub_FireFlag = 0;
    sw_XSignal = (T_SWORD) (((T_SLONG)uw_Sensor1_lms - ZERO_POINT) * SENSOR_X_FACT) >> 5);
    sw_YSignal = (T_SWORD) (((T_SLONG)uw_Sensor2_lms - ZERO_POINT) * SENSOR_Y_FACT) >> 5);
    if ( sw_XSignal > 187)
    {
        (*puw_MainSignalCounter)++;
    }
    else
    {
        (*puw_MainSignalCounter) = 0;
    }
    if ( sw_YSignal > 46)
    {
        (*puw_PlausibilityCounter)++;
    }
    else
    {
        (*puw_PlausibilityCounter) = 0;
    }
    if (((*puw_PlausibilityCounter) >= 5) && ((*puw_MainSignalCounter) >= 5))
    {
        ub_FireFlag = 1;
    }
    return(ub_FireFlag);
}
```

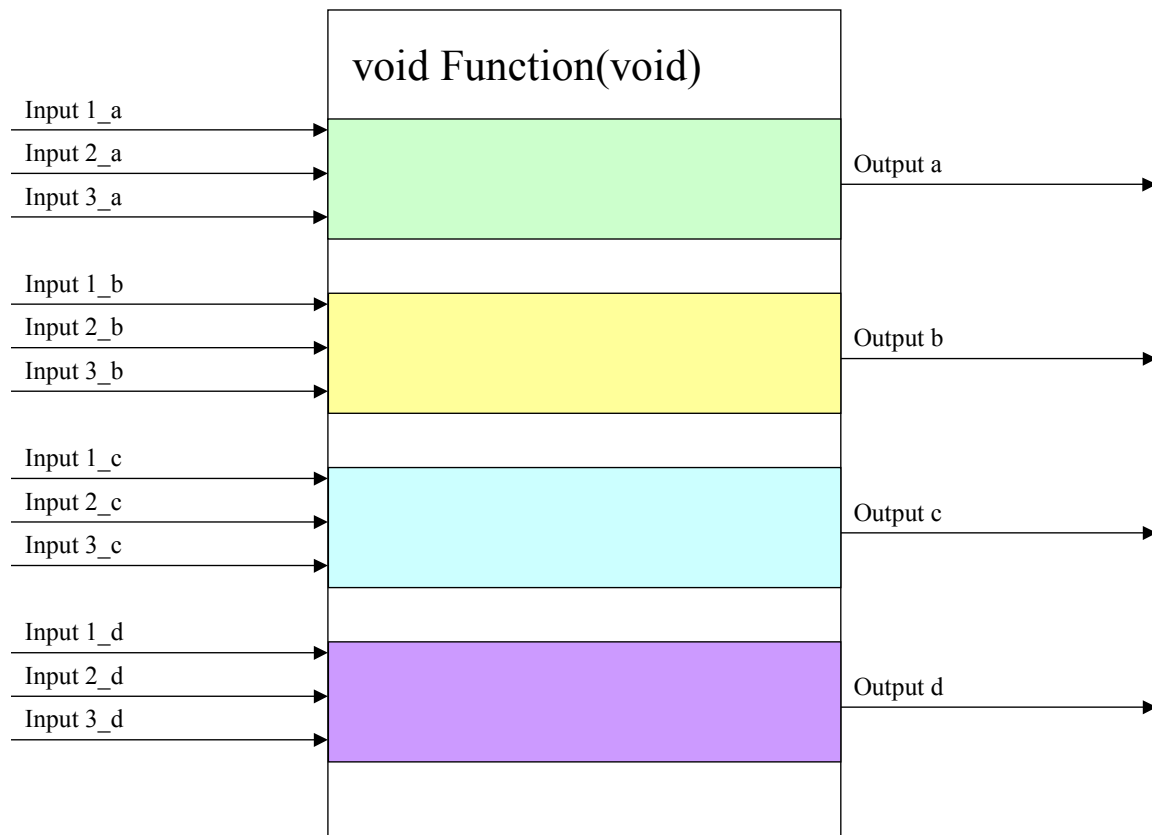
The change of the code lead only to minor modifications. However, everything which goes in or out of the function is now visualized in the definition of the function. The return value

contains the only functional relevant output of the function which is the Fire Flag. The counters are handed over as pointer. This way they can be read and written by the function. The two sensor values which are also in the pass parameter list are defined as "const". This way they can not be modified in the function. On the first glance this makes no sense, because they will be handed over to the function in CPU registers which are invalid after the exit of the function. Thus writing to these values is not possible anyway. However by defining them const everybody who reads the function will see on the first glance that this is a "read only" input. Note that this is only one possible design solution and may not be the best one because only the interface is considered here, e.g. without seeking for a better option for the counters in the function.

#### **1.3.4. Global Variables lead to a problematic Data Flow and Module Split Up**

I think so far the examples made clear that there is the co-existence of two main problems in an ad-hoc design. The first is that functions are declared "void" and the second is that since the pass parameter interface is not used the "interface" is implemented via global variables in these "void" functions. Consequently the function "interfaces" and the contents of the functions are too big and very often tailored badly. The following discussion about design problems can not so easily be seen in the example I used, since it is too small to be liable to this kind of problem. However, the further explanations I give should be clear enough to show that there might be big problems of this kind in larger programs.

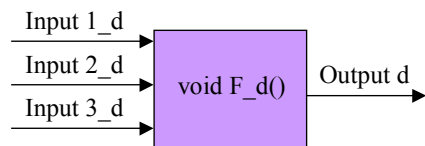
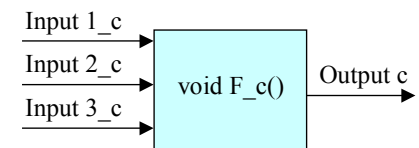
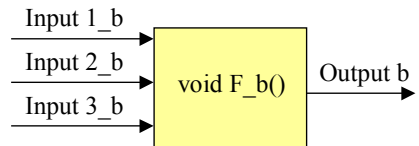
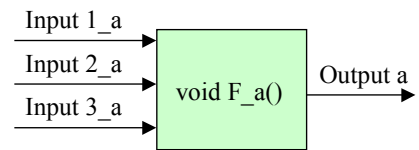
One of these design sins is the bad "cut" of modules and functions. In designs using void - void functions and global variables I came across functions which contained sequences of completely independent operational blocks which each worked on an own set of data of 3 inputs and one output. Each of these blocks could have been taken out into a separate function having a pass parameter list of 3 values and a return value. However in the ad-hoc design the function had 12 inputs and 4 outputs. Would the programmer have simply used the pass parameter interface he would have seen the awful design and split up the function into individual ones.



The split function below looks much better now. Each sub-function can be tested individually. Each of them only has 3 inputs and one output. In case the actual operation in the functions is the same, a further optimization would be to make only one subfunction and use pass parameters and the return parameter to interface to the one function. This way even ROM can be saved.

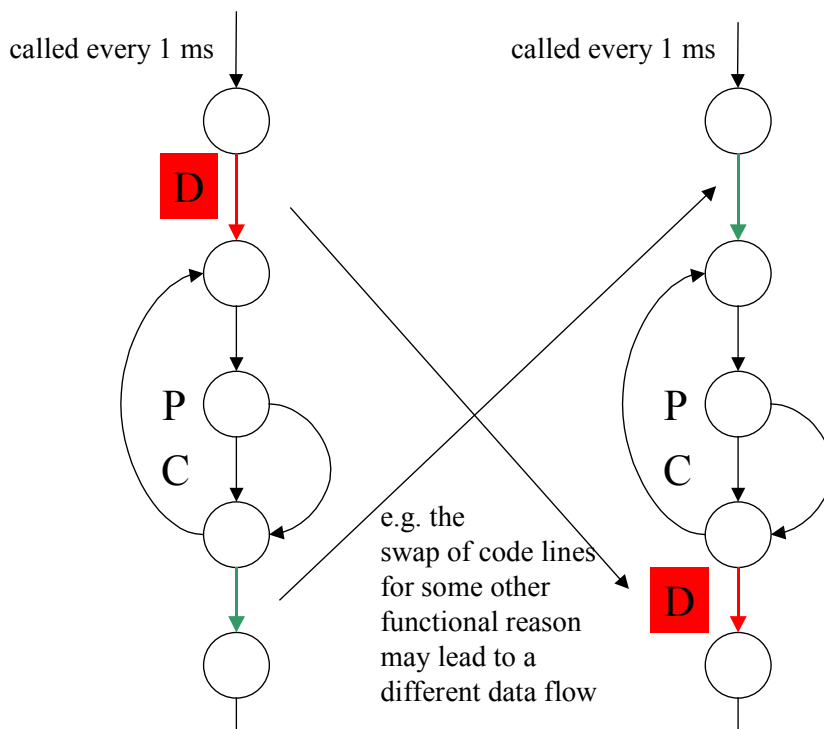
## void Function(void)

```
F_a();
F_b();
F_c();
F_d();
```



Besides the bad module split which leads to bad maintainability and testability some more serious problems may arise which really influence the functionality of the software or even constitute an error. These problems relate to the data flow in the system. To evaluate the data flow you always look at an individual variable and see where it is defined (a variable is created or a value is assigned to it) = **D**, where it is predicated (a variable is used in a compare) = **P**, where a variable is used in a calculation = **C** or where it is killed (made invalid e.g. by leaving the function) = **K**. This syntax of describing a data flow originates in software testing methods for data flow testing, however it is helpful at this place to describe the possible design problem.

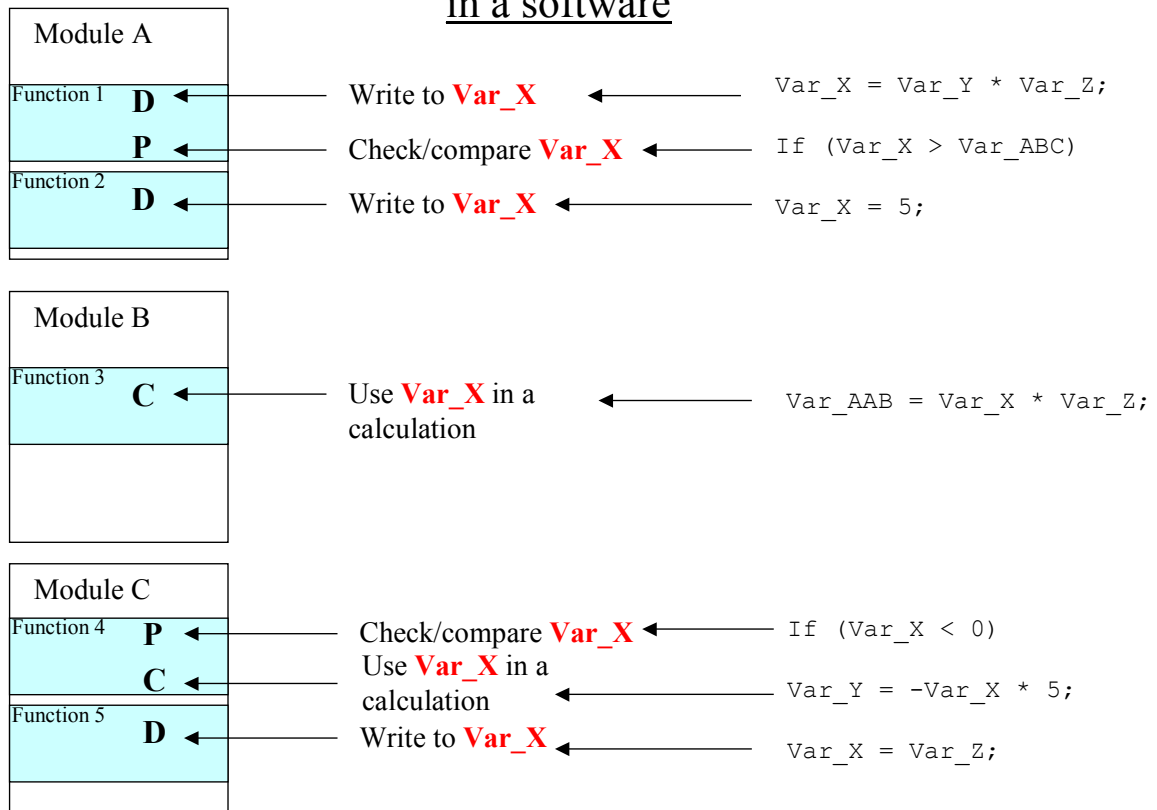
The use of global variables tempts to read them and write to them without considering the sequence of usage with the necessary diligence. Especially if the control flow of a program is changed, the underlying data flow may end up in a complete mess. Control flows in programs may be changed for various reasons throughout the lifetime of a project. This may be runtime optimizations for operating system tasks e.g. to distribute the load more equal to various tasks. Another reason may be timing or functionality improvements, e.g. to change the sequence of the setting of port pins. However with all these activities you may end up with unwanted effects. On the one hand you achieve the change of a port pin setting sequence, but since the function contains also other things, which do not necessarily relate to the port pin sequence their data flow may be changed in an unwanted way.



In the above example a small change to the code lead to a problem in the data flow. In the old version the data of a variable was aquired first and then processed in a predicate and a calculation within the same 1ms operating system cycle. After the update, however the data is aquired after it is used. Since all data are global, there will be no static problem. I.e. everything will compile and link just as before. However, from a dynamic point of view the data which is processed in a predicate and a calculation will be always 1ms old and not fresh. Depending on the application this may lead to a loss of performance or strange functional behaviour.

Further the use of global variables has also a big impact on testing. Especially the so called data flow testing will be much more difficult. The same negative impact of global variables can be seen for the maintenance of a software. The overall behaviour and data usage is hard to comprehend. The following pictures gives an impression of the problem:

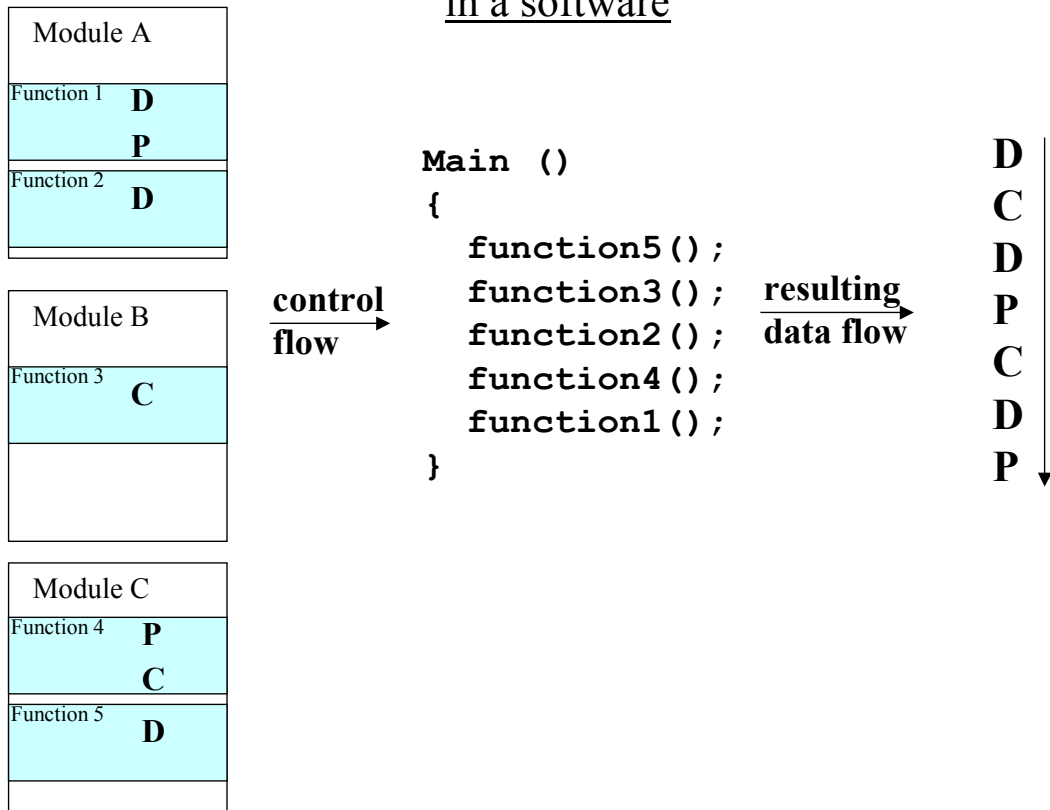
## Possible use of a global variable in a software



Data Flow Test Notation

The above picture shows how the same variable may be used at multiple places in the software. This may be different modules and different functions within the modules.

Possible use of a global variable  
in a software



This picture shows now how the control flow is responsible for the actual data flow. In order to comprehend the usage of a variable it is first of all necessary to detect in which modules and functions it is used. Next you have to comprehend the control flow with all it's possible conditions and branches before you can determine what is done to the data. This procedure has to be performed over and over again for each variable which is used, as you want to do maintenance on this kind of software. Very easily "data usage errors" slip in and may not be detected easiliy.

For testing this kind of software is simply a nightmare. You never can test a module "stand alone" because multiple modules may be involved in modifying and using data. The next hurdle is that data flow may be dependent on differing conditions across multiple modules. Further, regression tests may be a problem. As you can see, small changes in the control flow may lead to big changes in the data flow. This means that regression testing for the data flow is hardly possible. It always should be re-inspected manually.



Reasons for a problematic design are:

- The use of global variables will lead to a data flow which is scattered over multiple functions and modules. This means that maintainability and testability is very poor. Small modifications lead to a different data flow and possible data error, especially resulting in loss of system performance or functional errors.

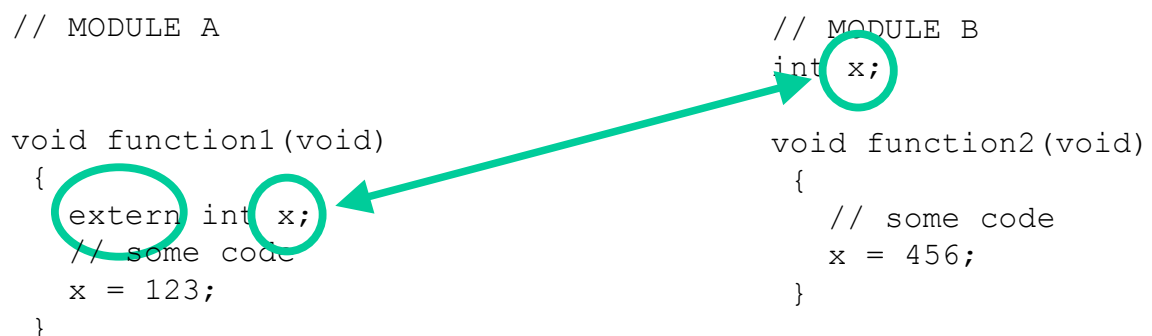
**Possible Solution:** As already mentioned the use of the technical interface of the programming language will solve most of the problems. Further the use of global variables has to be prohibited completely and an object oriented design should be implemented.

### 1.3.5. Other possible Pitfalls of Global Variables

#### 1.3.5.1. Unwanted Cross Influences

In a development setup with multiple programmers and a bigger amount of source code modules the danger is at hand, that a programmer accesses a global variable of another program component by simply referencing it as "extern" as it can be seen in the following picture.

```
// MODULE A                                     // MODULE B
void function1(void)                             int x;
{
  extern int x;                                  void function2(void)
  // some code                                   {
  x = 123;                                       // some code
}                                                 x = 456;
}
```

 A diagram illustrating unwanted cross influences. It shows two code snippets side-by-side. The left snippet is for 'MODULE A' and shows a function 'function1' that declares 'extern int x;' and assigns 'x = 123;'. The right snippet is for 'MODULE B' and shows a function 'function2' that declares 'int x;' and assigns 'x = 456;'. A green arrow points from the 'int x;' declaration in Module B to the 'extern int x;' declaration in Module A, indicating that Module A is accessing the variable 'x' defined in Module B.

This will lead to the following possible problems:

1. The global variable may be accessed for writing by an "outside" program component. The programmer of the Module B may not expect that anybody will write to "his" variable. However he has no possibility to avoid this. Thus the data flow might end up in a complete mess and errors may occur which are only dynamically detectable, i.e. in integration or system tests.
2. The programmer of the Module A who accesses the global variable of Module B may experience some surprises if the programmer of Module B will change the general properties of "his" global variable. If he changes the name, the compiler will give you an error, but if he changes e.g. the resolution or range of the contained data the programmer of Module A will end up with a functional problem which is only dynamically detectable.

So far the cross influences were due to deliberate modifications of the variable properties. However there are also possibilities of accidental cross influences. In many systems, especially those which have to rely on global variables for tool or simulation access, all global variables are defined in a single module. Additionally there will be a header file which declares all these variables as "extern". This header file is then included in all modules so that everybody can just use each variable without even having a need to make explicit "extern" declarations of a subset of these variables in particular modules. If this design option is paired with a bad naming convention you may end up in accessing wrong variables by typing errors. Look at the following code line and try to see the problem:

```
sw_XSignal = (T_SWORD) (((T_SLONG)uw_Sensor1_lms - ZERO_POINT) * SENSOR_X_FACT) >> 5);
```

What appears to be a "1" is in reality a lower case "l".

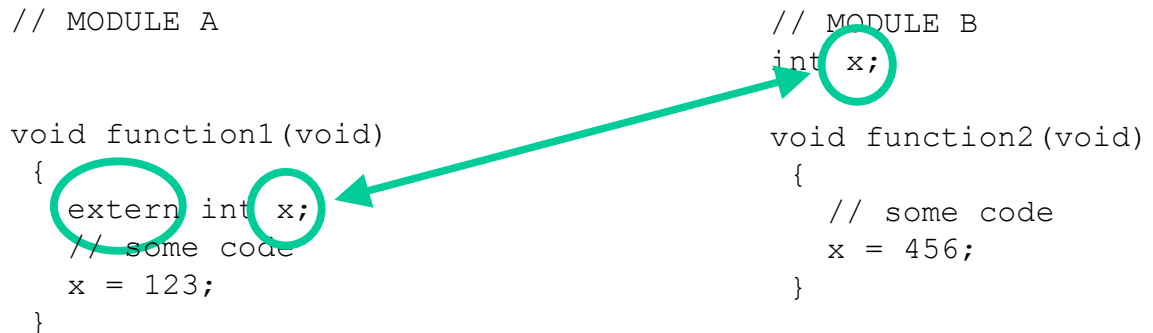
```
sw_XSignal = (T_SWORD) (((T_SLONG)uw_Sensor1_lms - ZERO_POINT) * SENSOR_X_FACT) >> 5);
```

In fact I came across software where variables were numbered and called "var1, var2, var3, etc." additionally there were variables called "varl" for low and "varh" for high. In this setup a typing error led to a strange functional error of minor kind. The variable "varl" was accessed instead of "var1" and gave strange unsmoothness to some upshifts in an automatic transmission. Nothing serious, but enough to be noticed and enough to require further attention. The focus was not on debugging this error because it was believed that the functional misbehaviour was due to a bad data calibration and all that is needed was a correct setting of the thresholds and timers to gain a smooth functionality. Only weeks later, after data calibration gave no success, the typing error was detected and fixed. Similar confusion can happen if you use "o" (lower case o), "O" (upper case o) and "0" (zero) in variable names.

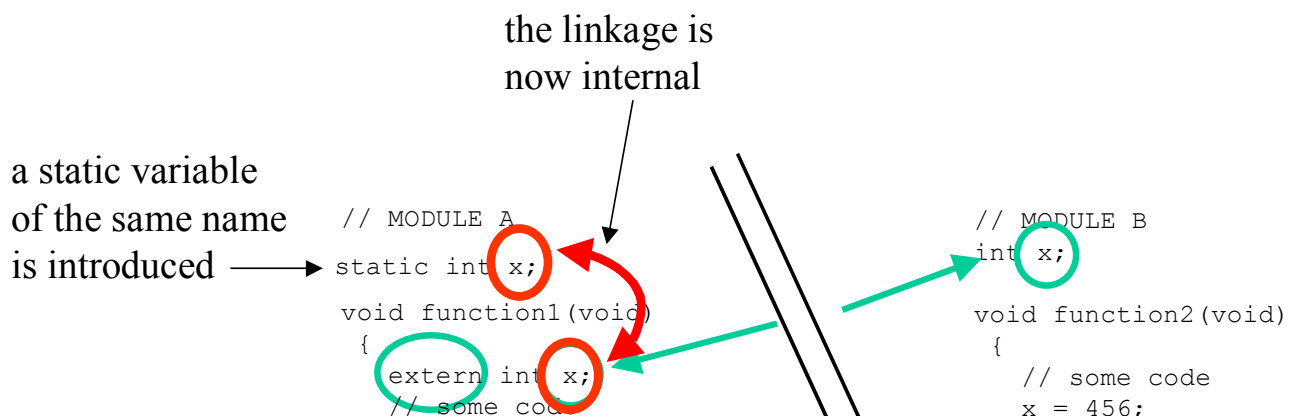
### 1.3.5.2. Change of Linkage

A special problem can occur due to the rules for linkage in the ISO C-Standard. The standard states that the linkage of variables declared as "extern" is prioritized to be first of all "internal"

and only secondly "external". This means that if there is an "extern" declaration of a variable in a module without the definition being in the same module the linker has to assign any other external variable i.e. of another module to be linked, as it can be seen in the following picture:



However, if a definition of a variable with a matching name is performed within the module the linkage will be performed internal inside the same module. Thus by a maintenance activity, where an additional variable is introduced, which accidentally has the same as an already extern referenced variable, the linkage will be changed as it can be seen in the following picture:



Also in this case there will be no warning or compiler error. This is a completely valid issue, but it will result in an error which can be only dynamically detected.



Reasons for a problematic design are:

- The use of global variables may have some additional pitfalls. Cross influences may occur by deliberate access across modules, by accidental mistyping of variable names or by the change of linkage according to the ISO C rules.

**Possible Solution:** As already mentioned avoid the use of global variables completely and use the technical interface of the programming language. This will solve most of the problems. Further you should define a clear naming convention to differentiate more clearly between variable types in order to reduce accidental errors.

### 1.3.6. The use of Basic Data Types

Finally I want to point out another design problem which is related to the use of the standard or basic data types in C. Many programmers have the programming style to define their variables with basic data types such as:

```
int MyVar; unsigned char MySecondVar;
```

This generates also a problem of portability.

First of all most automatic code checkers can not work on the basic data types and require a general redefinition of all of the types.

Secondly some of the data types have a different meaning depending on the compiler or CPU. For example there might be a difference between **char** and **signed char**. For some compilers this is the same, like **int** and **signed int** are the same. For other compilers **char** is a printable ASCII character and **signed char** is an integer number. If you directly use the data type **char** all over your code this may not be portable.

Thirdly the bit size of variables depends on the implementation (compiler or CPU). For example the data type **int** is a 16bit value for the C164 compiler. For a 32bit ARM-7 CPU as e.g. the TMS470 **int** is a 32bit value. If you now think that on a 8bit 8051 CPU **int** will be a 8bit value, you are wrong. **int** is a 16bit value in this environment as well. Due to this confusing situation it is highly recommendable to redefine your basic data types and use the definitions instead of the plain data types. This will keep your code portable, because you can easily adapt to other underlying basic data types at a single place. An example for a redefinition can be seen in the following lines:

```
typedef unsigned char      T_UBYTE;    // value range: 0.....255
typedef signed char        T_SBYTE;    // value range: -128.....127
typedef unsigned short int T_UWORD;    // value range: 0.....65535
typedef short              T_SWORD;    // value range: -32768.....32767
typedef unsigned long int  T_ULONG;    // value range: 0.....4294967294
typedef long int           T_SLONG;    // value range: -2147483648.....2147483647
typedef char               T_CHAR;     // value range: -128.....127
```